

# SUNMOS for the Intel Paragon

## A Brief User's Guide<sup>\*†</sup>

Arthur B. Maccabe  
University of New Mexico  
Albuquerque, NM 87131  
maccabe@cs.unm.edu

Kevin S. McCurley  
Sandia National Laboratories  
Albuquerque, NM 87185-1109  
mccurley@cs.sandia.gov

Rolf Riesen  
Sandia National Laboratories  
Albuquerque, NM 87185-1109  
rolf@cs.sandia.gov

Stephen R. Wheat  
Sandia National Laboratories  
Albuquerque, NM 87185-1109  
srwheat@cs.sandia.gov

Last revised October 20, 1994

## 1 Background

SUNMOS is an acronym for **S**andia/**U**NM **O**perating **S**ystem. It was originally developed for the nCUBE-2 MIMD supercomputer between January and December of 1991. Between April and August of 1993, SUNMOS was ported to the Intel Paragon. This document provides a quick overview of how to compile and run jobs using the SUNMOS environment on the Paragon.

The primary goal of SUNMOS is to provide high performance message passing and process support while consuming a minimal amount of memory. As an example of its capabilities, SUNMOS Release 1.4 occupies approximately 240K of memory on a Paragon node, and is able to send messages at bandwidths of 165 megabytes per second with latencies as low as 42 microseconds using Intel NX calls. By contrast, Release 1.2 of OSF/1 for the Paragon occupies approximately 7 megabytes of memory on a node, has a peak bandwidth of 65 megabytes per second, and latencies as low as 42 microseconds (the communication numbers numbers are reported elsewhere in these proceedings [1]).

A Paragon running SUNMOS will use OSF in the .service and .io partitions, but will have SUNMOS loaded on all or some of the compute nodes in place of OSF. Compute nodes running SUNMOS do not appear in the .compute partition. The number and configuration of SUNMOS and OSF compute nodes is decided at boot time; see [3] for further details.

Through emulation libraries, SUNMOS currently supports many of the nCUBE message passing routines (e.g., `nread` and `nwrite`) and many of the NX message passing routines (e.g., `csend`, `isend`, `crecv`, and `irecv`). In addition, the standard SUNMOS library supports the C standard I/O library and the FORTRAN I/O library. As a consequence, many nCUBE and Intel NX codes will run under SUNMOS without modification.

---

<sup>\*</sup>This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

<sup>†</sup>This document applies to SUNMOS release 1.6.3. You can obtain the latest version of SUNMOS documentation via anonymous ftp from `ftp.cs.sandia.gov` in the directory `pub/sunmos/doc`.

A separate document [2] describes the differences between standard Intel NX routines and the SUNMOS emulation library. Another document [3] describes the installation procedure. In addition to these documents, there are man pages for the following SUNMOS commands and library routines:

```
yod fyod fservers create_yod_config getcomm getpcb showmesh _nsend/_recv
```

All of these are available by anonymous ftp from `ftp.cs.sandia.gov` in `pub/sunmos/doc`. If you need additional support or have further questions, send email to `sunmos-support@cs.unm.edu`.

## 2 Compiling SUNMOS applications

At Sandia, SUNMOS executables are generated on Sun workstations using the Intel supplied cross compilers. While it is possible to compile application programs on the service nodes on a Paragon, we do not recommend this practice.

The SUNMOS distribution comes with three shell scripts that can be used to compile programs written in C, C++, and FORTRAN. These scripts are called `sicc`, `siCC`, and `sif77`, respectively. These scripts invoke the appropriate cross compilers and link the application with the SUNMOS libraries.

These shell scripts, other SUNMOS utilities, and other useful files are located in a single directory tree. Check with your local system administrator for the location of this directory at your site.

The `sunmos` directory contains several subdirectories:

`current/bin` contains executables for the SUNMOS utilities. You will want this directory in your path.

`current/man` contains documentation on SUNMOS utilities. You may want to add this directory to your manpath.

`current/include` contains header files for the SUNMOS libraries. This directory is automatically searched when you use the SUNMOS compiler scripts, so you shouldn't need to reference this directory explicitly.

`current/lib` contains the SUNMOS libraries. This directory is automatically searched when you use the SUNMOS compiler scripts, so you shouldn't need to reference this directory explicitly.

With the exception of the `-nx` flag, all compiler flags are treated the same way under SUNMOS as under OSF, so there should be minimal changes required to makefiles. The `-nx` flag is used under OSF to link OSF libraries for compute nodes, and *should not* be used under SUNMOS.

## 3 Running applications—yod

The previous section described a cross-compilation environment on a Sun workstation. There are several utilities on the Paragon itself located in the directory `/sunmos/bin`. The most important ones are:

`yod` – the generic host node program; handles mesh allocation, program load and execution, and file I/O. For users familiar with the nCUBE, this corresponds to the `xnc` command. For users familiar with the Intel Delta, this corresponds to the `mexec` command.

**showmesh** – shows the current mesh allocation. An alternate tool called **showparts** is available via anonymous ftp from `ftp.cs.sandia.gov` in the directory `pub/paragon-contrib`.

**getcomm** – displays the communication buffers for a stopped process (useful in debugging).

**getpcb** – displays the process control block for a stopped process (useful in debugging).

In the remainder of this section we describe how **yod** is used to allocate, load and support the execution of application programs. In the next section we describe how to use **getcomm** and **getpcb** during application debugging. Once you are familiar with the basic operation of these utilities, you should consult the man pages for further details.

The **yod** program runs in the service partition, and handles all requests from the SUNMOS compute nodes that it controls (much like the proxy process under OSF). A special message passing module written by Intel allows communication between **yod** and SUNMOS running on compute nodes.

One important point about **yod**: when you abort a job under control of **yod**, you should be very careful about how you kill **yod**. The correct way to do this is to issue a control-C to the **yod**, or to type `kill <pid>` from another shell. *Do not* use `kill -9` or `kill -KILL` to kill a **yod** process. Doing so will leave your nodes allocated, and prevent future runs on those nodes until the machine is rebooted (or until `create_yod_config` is run again). If you run `create_yod_config` while there are running jobs, these runs are likely to be corrupted, so avoid running it unnecessarily.

### 3.1 An example

Before we consider the command line options supported by **yod**, it is instructive to consider a simple command line:

```
% yod -sz 8 a.out 100 200
```

This command line instructs **yod** to allocate 8 nodes from the SUNMOS nodes and load the application `a.out` on all 8 of these nodes. Any arguments after the name of the file containing the executable are passed to the application as command line arguments. In this case, each of the 8 application processes is provided with the command line arguments ‘100’ and ‘200’.

### 3.2 Node allocation

When you load and execute an application program, **yod** allocates the nodes that the application uses. When the application completes, **yod** reclaims the nodes for use by other applications.

There are three command line options that control the algorithm **yod** uses to allocate nodes: `-size`, `-allocation`, and `-base`.

The `-size` option can be abbreviated as `-sz` (as shown in the previous example). This option controls the number of nodes allocated for the application. If you don’t specify a size, the default is to allocate a single SUNMOS node. The size can be specified as an integer (e.g., 8) or a rectangle (e.g., 2x4). When the size is specified as a rectangle, the height is given first.

When you just specify the number of nodes to be allocated, **yod** first tries to allocate a rectangular region of the mesh, trying to keep the region as square as possible. If **yod** is unable to allocate a rectangular region, it will try to scatter the application processes throughout the mesh (still trying to keep them reasonably close together). If you specify the size as a rectangle, **yod** will only consider rectangular regions of the specified shape.

Using the `-allocation` option, you can control the allocations that `yod` will consider. `Yod` recognizes three allocation modes: strict, lax, and any. When the mode is *strict*, `yod` will only attempt contiguous allocations, i.e., the processes in your application will not span nodes used by the processes in another application. This mode is useful if you are concerned that another application might saturate a communication channel and interfere with the communication in your application.

When the allocation mode is *lax*, `yod` will still attempt to perform a strict allocation. If that fails, `yod` will then try to allocate nodes in a rectangular shape, spanning rows and columns with other application processes. (This is the default allocation mode when you specify the size as a rectangle.)

When the allocation mode is *any*, `yod` will first attempt a strict allocation. If that fails, `yod` will then attempt a lax allocation. If that fails, `yod` will then try to find the desired number of nodes anywhere in the mesh. (This is the default allocation mode when you specify the size as a single number.)

The `-base` option controls the starting position for an allocation in the mesh. This option is useful when you need to take advantage of differences in the nodes (e.g., different memory sizes).

### 3.3 Controlling memory allocation

When an application process is loaded, SUNMOS allocates memory for five distinct regions: code (text), static data, communication, stack, and heap. The code region is always just large enough to hold the code for the process. Similarly, the static data region is always just large enough to hold the static data for the process. By default, SUNMOS allocates 256K bytes for the communication and stack regions. After it has allocated memory for the code, communication, and stack regions, SUNMOS, by default allocates the remainder of the application memory (i.e., the memory that is not used by SUNMOS itself) for the application's heap region.

You can directly control the amount of memory allocated for the communication, stack, and heap regions using the `-comm`, `-stack`, and `-heap` command line options of `yod`. Each of these options takes a single number, the number of bytes to allocate for the specified region. As an example, if you use `-comm 1000000`, then SUNMOS will allocate 1,000,000 bytes of storage on each node for message buffers.

The communication region is used to buffer messages that have arrived at the receiving node, but not yet requested by the application process. If you allocate too little space for the communication region, SUNMOS will abort the application processes as soon as a message arrives when there is not sufficient space in communication region to hold the message.

The stack region is used to hold local variables. If you use large automatic arrays, you may exceed the default stack size. If your application exceeds the size allocated for the stack region, the results are unpredictable; however, you will most likely get a data access fault during execution.

The heap region is used to hold the dynamic space used by an application process. In C programs, this space is accessed using the standard library routines: malloc, calloc, and free.

## 4 Debugging (such as it is)

There is currently no support for a debugger under SUNMOS (aarrgh!). We recommend debugging under OSF, which has integrated debugger support. Support for a debugger may be added in the future. At present it is only possible to use ipd to decide where a program has hung, but not to inspect variables (more on this below in the discussion of node fault dumps).

As a poor man's substitute for a debugger, you can observe the front panel lights and use the `getcomm` and `getpcb` utilities to find the current state of a hung process.

## 4.1 The lights

The lights on the front of the machine can sometimes be used to diagnose what went wrong. Each node has 6 lights associated with it: one red light, and five bar lights. The bar lights on each node will be referred to as numbered 1–5, counting from the top.

- when a node is not running a user process, the lights repeatedly blink in the pattern 1—2—3—4—5—4—3—2—1.
- when the primary processor on a node has faulted, the lights repeatedly blink the pattern: 3—2,4—1,5—2,4—3.
- when the coprocessor on a node faults (and the primary processor does not) light # 3 comes on and stays on.
- when the processor is running in user mode, light # 1 comes on and stays on. This includes the case when a user is blocked waiting for a message.
- when a node is in system mode but hung receiving a message, light # 4 is on. This is usually accompanied by the red light on the node being on, and will usually require a reboot of the machine. This indicates an OS failure and should be investigated further if it happens.
- when a node is in system mode but hung sending a message, light # 5 is on. This is usually accompanied by the red light on the node being on, and will usually require a reboot of the machine. This indicates an OS failure and should be investigated further if it happens.

## 4.2 Inspecting the message queues of hung jobs

The `getcomm` utility can be used to display the list of unreceived messages and posted receives. A sample output for `getcomm` is given in Figure 1. This shows the output from a one node of a program that was run on physical nodes 12 and 13, running to a certain point and hanging.

The output from `getcomm` shows that node 12 (using physical node numbering) is currently blocked waiting for a message of type 54 and length 0 from any source (0x0000ffff), and has received a message from logical node 1 (physical node 13) of type 55 and length 2 bytes in the system buffer.

The utility `getpcb` is a bit more obscure—see the man page. One use is to decide what state a node is in. The SUNMOS kernel does not queue outgoing messages. If it is interrupted by a `getcomm` request while sending a message, the request will be ignored. When this happens `getcomm` will hang, waiting for the kernel reply.

## 4.3 Deciphering node fault dumps

When a SUNMOS node process faults (yes, it can happen), it displays some very crude register, stack, and process information that can be used to diagnose the failure (those of you old enough to remember the 1960's will feel right at home).

Consider the example program given in Figure 2. When `func2()` is called, it will produce a floating point overflow, resulting in a fault on the node (recall that SUNMOS does not handle floating point exceptions). When this happens the node dump of Figure 3 is produced.

```

% getcomm -n 12

Comm Buff Analysis for node 12 (logical d0)
-----
User has not processed the following message(s) yet:
! len= 2, type= 55 (0x00000037), src= 13 (0x0000000d), src_pid 1
!           logical src= 1 (0x00000001)
!           dst= 12 (0x0000000c), dst_pid 1

===== time stamp = 0x0000017a

User is waiting the following message(s):
? len= 0, type= 54 (0x00000036), src 65535 (0x0000ffff), pid 65535 (0x0000ffff)
?           time stamp = 0x0000017e, msg body= 0xf0f7b9d8

```

Figure 1: Sample output from getcomm.

```

#include <stdio.h>          /* LINE NUMBER */
#include <math.h>            /*      2      */

void
func1(x)
float x;
{
    func2(x);
}

func2(x)
float x;
{
    float y = MAXFLOAT;
    fprintf(stderr, "Hello World - %3d\n", mynode());
    x += y;
    exit(0);
}

main()
{
    float x = MAXFLOAT;
    func1(x);
}

```

Figure 2: A sample program that will fault under SUNMOS.

```

Hello World - 0
NODE 0x23 (logical 0): proc 0x0, psr 0x410a0, epsr 0x61080402, fsr 0x142440a1
    pc(log) 0x10234, pc(phys) 0xf0f10234, instr 0x2c720012, sp 0x5f8fd40
    text_base= 0x00010000, text_len= 86016, text_end= 0x00025000
    data_base= 0x04014000, data_len= 28672, data_end= 0x0401b000
    comm_base= 0xf0f6c000, comm_len= 262144, comm_end= 0xf0fac000
    heap_base= 0x04800000, heap_len= 15794176, heap_end= 0x05710000
    stk_base1= 0x05f50000, stk_len1= 262144, stk_end= 0x05f90000
    stk_base2= 0x00000000, stk_len2= 0, stk_end= 0x00000000
    stk_base3= 0x00000000, stk_len3= 0, stk_end= 0x00000000
FAULT TYPE: Floating point Trap
                    Adder overflow bit set in fsr

r0: 0x00000000, r1: 0x00010228, r2: 0x05f8fd40, r3: 0x05f8fd50
r4: 0x05709b2c, r5: 0x000000d7, r6: 0x00000000, r7: 0xfffffec0
r8: 0xffffffff, r9: 0x00000001, r10: 0x0401a0d0, r11: 0x0401a0e4
r12: 0x00000017, r13: 0x05f8feec, r14: 0x04018540, r15: 0x05f8feee
r16: 0x00000012, r17: 0x00015808, r18: 0x000401a0, r19: 0x61080402
r20: 0x350000a1, r21: 0xf7fecaa60, r22: 0x00000000, r23: 0x00000000
r24: 0x00000000, r25: 0x00000000, r26: 0x00000000, r27: 0x00002000
r28: 0x00000000, r29: 0x00000086, r30: 0x00000086, r31: 0x05f8fd40

f0: 0x00000000, f1: 0x00000000, f2: 0x00000000, f3: 0x00000000
f4: 0x00000000, f5: 0x00000000, f6: 0x00000000, f7: 0x00000000
f8: 0x7f7fffff, f9: 0x47efffff, f10: 0x00000000, f11: 0x00000000
f12: 0x00000000, f13: 0x00000000, f14: 0x00000000, f15: 0x00000000
f16: 0x7f7fffff, f17: 0x7f7fffff, f18: 0x7fffffff, f19: 0x00230000
f20: 0x00000001, f21: 0xfffffffef3, f22: 0x00000004, f23: 0x00000001
f24: 0x00000000, f25: 0x3fb99996, f26: 0x40000000, f27: 0x3ff00002
f28: 0x00000000, f29: 0x43300000, f30: 0x00000000, f31: 0x3fa99996

Stack Trace: Fault instr 0x10230
    Addr(1): 0x101bc
    Addr(2): 0x10280

```

Figure 3: Sample output from a node that faulted under SUNMOS. Included are the contents of the floating point and integer registers, the type of fault that occurred, and stack information that can be used with ipd to discover the location of the fault. Earlier versions produced somewhat different output.

For this example, the interesting part is given at the bottom in the stack trace, where the address of the faulty instruction is given. If the program was compiled with the `-g` option, then you should now start `ipd` on a service node. At the prompt, type the following commands:

```
ipd > load a.out
(host) > disa 0x10228,10
```

This will load the executable (called `a.out` here), and disassemble the section of code beginning two instructions (8 bytes) ahead of where the fault occurred (`0x10228` is 8 bytes ahead of `0x10230`, which was where the fault occurred). The output from the last command is:

```
(host) > disa 0x10228,10
***** (host) *****
fault.c{}func2()#16
00010228: 24700012 fld.l      16(fp), f16
0001022c: 2471ffffe fld.l      -4(fp), f17
00010230: 4a128830 fadd.ss    f17, f16, f18
00010234: 2c720012 fst.l      f18, 16(fp)
fault.c{}func2()#17
00010238: 6c0011e1 call       +0x4788 <0x149c0>
0001023c: a0100000 mov        r0, r16
fault.c{}func2()#18
00010240: 947f0060 adds      96, fp, r31
00010244: 14610005 ld.l      4(fp), r1
00010248: 14630001 ld.l      0(fp), fp
0001024c: 40000800 bri       r1
```

This tells you that the code at line 16 of file `fault.c` generated an instruction `fadd.ss` at address `00010230`, which is precisely where the error occurred. Due to complications in the instruction set, the exact address of the faulting instruction may be off by one instruction (four bytes).

In case you need to find out how the code arrived at this instruction, the stack trace tells you the addresses of the function calls that were made to get to this point. In the example, the address `0x101bc` from the stack trace can be disassembled to reveal that this is the address of the call to `func2()` from `func1()`, and `0x10280` is the address of the call to `func1()` within `main()`. If you use the options `-Manno` and `-S` to `sicc` you will also get an annotated assembler listing that can be useful.

You might wonder what happens if you forgot to compile with the `-g` option, and the fault occurred after an overnight run. In this case, if you at least have the `-Mframe` option, then you will still get the stack frame information about what functions were called, but in `ipd` you will only have address offsets (in bytes) into the function rather than line number information. Optimization may also produce less information (or hard to understand information).

This is not fun—or particularly efficient—but sometimes better than nothing. If you’re faint of heart, then try the old 1970’s way of inserting print statements.

## 5 I/O and fyod

The `fyod` program provides scalable file I/O service to SUNMOS nodes. If this program is not used, then all I/O from a SUNMOS job is funneled through the `yod` process on the service partition

of the Paragon. While this is simple and adequate for small numbers of nodes or small amounts of I/O, it represents bottleneck in I/O bandwidth.

**fyod** is intended to help with this difficulty. Users can start an **fyod** process on the .io partition to handle requests to different directories in parallel. As an example, suppose the paragon has a 2 node .io partition with 2 RAID devices: `/raid/io_01` and `/raid/io_02`. The user could have these managed separately by typing

```
fyod -sz 2 -pn .io -dir /raid/io##
```

The use of `##` indicates that requests to open a file from a directory with `##` replaced by one of the strings “01” or “02” are directed to the appropriate I/O node. A similar paradigm exists in the NX library when a file is opened: a string of three or more `#` characters in a file name is replaced by the node number of the job opening them). To describe how the **fyod** program works, we start with an **fopen** request made by a SUNMOS node. This is sent to the **yod** process responsible for the job. **yod** then starts by looking up in a directory of file services. If an **fyod** service is found that handles the directory in which the file resides, the open request is sent to the appropriate **fyod**. This **fyod** opens the file and sends a response back to the SUNMOS node. Future I/O traffic for this file travels between the SUNMOS node and the I/O node directly, without intervention from the **yod** process. A user can display the directory of existing **fyod** services by typing the command **fservers**. At some point in the future, **fyod** will likely become part of the boot procedure, eliminating the need for users to manage their own I/O servers. Further details on **fyod** appear in the man page. **fyod** requires a large segment of wired memory on an I/O node, so it is not advisable to run more than one of these on a single I/O node. This will also result in a decrease in performance.

## 6 Advanced features of **yod**

### 6.1 Heterogeneous loads

A recently added feature to **yod** is the ability to load different executables on different nodes, all within a single application. The exact syntax for doing this should be contained in the man page, but roughly speaking you use **yod -F loadmap** to specify that loads are controlled by the contents of the file **loadmap**. For example, if on the command line you typed

```
yod -heap 500000 -base e0 -F loadfile
```

and the loadfile contained:

```
4x4
yod -heap 300000 -sz 2x2:0,0 prog1 arglist1
yod -stack 500000 -sz 2x4:2,0 prog2 arglist2
yod -sz 2x2:0,2 prog3 arglist3
```

then a 4x4 mesh would be tiled with programs 1,2, and 3 as follows:

1	1	3	3
1	1	3	3
2	2	2	2
2	2	2	2

The heap specification for `prog1` would override the specification on the command line, but all others would get the heap specification from the command line. Further details are available from the man page for **yod**.

## 6.2 Use of the second processor

The GP node of an Intel Paragon has two i860XP processors that share access to the memory (MP nodes will have even more). Originally Intel planned to use this processor as a communications coprocessor, with the goal of lower latency and ability to overlap computation and communication on a node. Under SUNMOS, there are currently three modes supported for use of the second processor:

**mode 0** affectionately called “heater mode”; where the second processor is inactive.

**mode 1** the second processor is used as a communication coprocessor. This results in significantly lower latencies for message passing, and allows better overlap between communication and computation.

**mode 2** the second processor can be used as an additional compute processor, with shared memory. This has been referred to by some as “SUNMOS turbo mode”.

Modes 1 and 2 will not work reliably with some early hardware, but seem to work reliably now on the Sandia hardware. These modes are selected through the `-proc` option to `yod`.

Mode 2 requires a user to specify some work to be done on the second processor via a function or subroutine call that takes a single integer argument. There are currently some restrictions on what can be called on the second processor. In particular, you cannot print from the coprocessor, send messages, or use certain system calls that use static variables (and hence are not reentrant). The two processors will share the heap, but have separate stacks (as a result, the use of the second processor will consume extra memory on a node). The performance improvement that is achieved using mode 2 depends primarily on the degree to which the processors use their memory caches. We have witnessed speedups as high as 95% for codes that reuse their caches very well. In particular, it was possible to achieve a speed higher than the rated peak of the Paragon using this method (!).

In the C language, the interface to the second processor in mode 2 is through a function `cop()`, whose prototype is:

```
cop(void (*f)(), volatile unsigned *flag,  
    void *arg);
```

The function `f` to be run on the second processor should have a prototype

```
void f(int arg);
```

In order to run `f(arg)` on the second processor, you would use statements like

```
volatile int flag=0;  
cop(f,&flag,&arg);
```

The main processor can then go off to do other work, and when the function `f` completes on the second processor, `flag` will be incremented. The main processor can later check for completion of `f` by inspecting the value of `flag` to see if it was incremented.

There is currently no interface to the second processor directly from Fortran, but this will be corrected in future versions of SUNMOS. For now, the following piece of C code can be used. In a file called `fcop.c`, put the following lines:

```
BAD TEXT LOAD: tmin=0x10023222 tmax=0x10012122 s30sum=1001
BAD DATA LOAD: dmin=0x10921213 dmax=0x0x110271 s30sum=1172
```

Figure 4: Error messages from a bad load.

```
static volatile iflag;
void fcop_(void (*addthem)(), int *j)
{
    iflag=0;
    cop(addthem,&iflag,j);
}
void fcopdone_(int *flag)
{
    *flag = iflag;
}
```

Compile `fcop.c` using the command

```
%sicc -c -O3 fcop.c
```

Then link the file `fcop.o` with the rest of your application. In order to call a subroutine

```
subroutine f(i)
```

on the second processor, you use the line

```
call fcop(f,i)
```

Be sure to declare that `f` is an external subroutine. You can later check for completion using the lines

```
iflag = 0
call fcopdone(iflag)
... other work on main processor
if (iflag .eq. 1) then
    ... f completed
```

The file `fcop.c` is currently in the directory `/home/u/mccurley/fcop` inside the `cs.sandia.gov` domain, along with a test Fortran program called `test.f`. It should also be on `ftp.cs.sandia.gov` in the directory `pub/paragon-contrib`.

## 7 Other issues

There has been a data corruption problem during the load in some of the early 32-megabyte nodes on the Sandia machine. If you observe this, it will produce an error message of the form shown in Figure 4. This informs you that the text or data segment of your loaded program produces an incorrect checksum, so that the program may give incorrect results. One way to protect against it is to specify a heap that fits in 16 megabytes when loading on 32 megabyte nodes.

The processor modes 1 and 2 are not sufficiently tested, and some bugs are known to exist on some hardware. The Sandia machine has had all of the node boards upgraded to fix the known bugs, and any new bugs should be reported to `sunmos-dev@cs.sandia.gov`. Other sites may observe problems with these modes if their hardware is an early version and has not been upgraded.

At present there is no document describing the limitations of the nCUBE emulation library, but some notable omissions are the lack of global operations such as `ndsumn()` and `nglobal()`. Broadcasting by giving a destination of -1 to `nwrite()` is also not currently supported.

Floating point exception handling for IEEE floating point arithmetic is currently not implemented in SUNMOS. This is a design decision dictated by the fact that such exception handling is done in software, is incredibly slow, occurs rarely, and would essentially double the size of the SUNMOS operating system.

Constructive feedback on this document is welcome. Send comments to `mccurley@cs.sandia.gov`. Send complaints to `/dev/null`.

## References

- [1] Bernard Traversat, Bill Nitzberg, and Sam Fineberg, Experience with SUNMOS on the Paragon XP/S-15, in ISUG-94.
- [2] Kevin S. McCurley, Intel NX compatibility under SUNMOS, Sandia National Laboratories Technical Report # 93-2618.
- [3] T. Mack Stallcup, Installation Instructions for SUNMOS.
- [4] man pages for yod, fyod, fservers, getpcb, getcomm, showmesh, and nsend/nrecv.